Journal of Universal Computer Science, vol. 10, no. 9 (2004), 1272-1293
submitted: 7/7/03, accepted: 5/4/04, appeared: 28/9/04 © J.UCS

A Formal Model of Forth Control Words in the Pi-Calculus

James F. Power
(National University of Ireland, Maynooth, Ireland
jpower@cs.may.ie)

David Sinclair
(Dublin City University, Ireland
David.Sinclair@computing.dcu.ie)

Abstract: In this paper we develop a formal specification of aspects of the Forth programming language. We describe the operation of the Forth compiler as it translates Forth control words, dealing in particular with the interpretation of immediate words during compilation. Our goal here is to provide a basis for the study of safety properties of embedded systems, many of which are constructed using Forth or Forth-like languages. To this end we construct a model of the Forth compiler in the π-calculus, and have simulated its execution by animating this model using the Pict programming language.
Key Words: Forth, pi-calculus, operational semantics
Category: D.3.1, D.3.3, F.3.2

1 Introduction

In this paper we seek to contribute to the study of stack-based languages and architectures by providing a model of the control structures used in the Forth programming language. Stack-based machines have a long history in programming language implementation [Koopman 1989], but lately have achieved increased prominence due to the widespread use of virtual architectures such as the Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR).

The remainder of this paper is structured as follows. Section 2 presents the background to our work, and briefly surveys stack-based languages, formal semantics and the π-calculus. Section 3 presents a high-level overview of the processes in our model. This section provides a context for Sections 4 and 5, which present the main aspects of the specification. Section 6 presents some examples of the use of our specification to translate some common Forth idioms. Finally, Section 7 concludes the paper.

2 Background and Related Work

The Forth programming language is relatively old in the context of high-level programming languages, dating from around 1970 [Moore and Leach 1970]. Its emphasis on high performance coupled with a small memory footprint has helped establish the language, particularly in relation to embedded microcontroller systems and similar industrial applications. Reflecting this, Forth has been standardised by both ANSI and the ISO [ISO 1997].

In this section we give an overview of stack-based machines and point to some of the strengths and weaknesses of existing implementations of this approach. In particular we highlight the importance of formal models in establishing safety properties, particularly in relation to Forth's control structures. We also present an overview of the π-calculus, the formalism used later in the paper to model the interaction between the processes in the Forth compiler.

2.1 Stack-Based Machines

In the extreme case a stack-based language will insist on all programming activities being performed directly on the stack, including expression evaluation, local variable storage, parameter passing and the return of result values from functions. More realistically, many such languages will mask these basic operations with a friendlier syntax.

Much of the original motivation for the use of stack-based machines in programming language translation was as an abstract "machine code". Such languages were low-level enough to allow straightforward translation to a given assembly language for a specific architecture, but yet sufficiently high-level to allow the compiler-writer ignore many implementation details, most particularly the number and nature of the target architecture's registers.

The development and increasing popularity of the Java programming language [Gosling et al. 1996] and the JVM [Lindholm and Yellin 1996] have sparked renewed interest in the pragmatics surrounding stack-based language design. The Java language technology typically involves a Java compiler that translates Java programs into bytecode, along with an interpreter/compiler (the JVM) that executes this code. However, such systems still have difficulty in competing with Forth-based applications in terms of speed and memory efficiency [Barr and Frank 1997], although the situation is improving.

One of the major features of the JVM is also one of its major overheads - the use of stack-safe code that can be statically verified. Coupled with garbage collection and other security features, the JVM is probably at the extreme end of "safe" virtual machines. A more pragmatic approach, exemplified by the .NET Common Language Runtime [ECMA-335 2001], is to explicitly identify "unverifiable code" sections where pointer manipulation and manual memory allocation can happen.

Forth combines a distinctive postfix stack-based programming approach with a remarkably economical syntax. While fundamentally similar to the JVM and CLR architectures, Forth differs from both the JVM and CLR in that it does not provide direct primitive support for classes and objects. However, Forth does provide a flexible yet structured approach to the implementation of flow of control that contrasts sharply with that of the JVM and CLR, and it is Forth's implementation of these structures that forms the focus of this paper.

Forth control structures are a unique combination of high-level structured concepts with the flexibility of low-level test-and-branch operation. Further, Forth is unusual in that it allows the programmer define new control words, providing for a variety of possible constructs. One disadvantage of this approach, however, is that these control words can increase the complexity of the code, and can cause unexpected side-effects if used incautiously.

Systems such as the JVM and CLR incur significant performance overheads by dealing with safety properties at run time, using a bytecode verifier. Indeed, one alternative to JVM bytecode, also designed to allow safe, mobile code, deliberately preserves high-level control structures for this reason [Franz 1998]. However, in industrial critical applications, and particularly with embedded systems, run-time failures are unacceptable, and considerable emphasis is placed on testing and static verification of the software.

2.2 Formal Semantics and the π-Calculus

In the following sections we present a formal model of the Forth compiler, highlighting the compilation of Forth control words. Providing formal definitions of programming languages is a well-established field, usually known as formal semantics (see e.g. [Watt 1991] for a survey) and, in this context, our definition would most likely be classed as an *operational semantics* of Forth.

However, the structure of the Forth compiler lends itself to a particular form of specification. Specifically, the operation of Forth is most often described in terms of the Forth engine's concurrent interactions with the control, data and return stacks, with the total effect being the parallel composition of these interactions. Thus the words in a Forth program become events that trigger changes in the processes representing the Forth stacks, along with other internal structures such as the dictionary.

This contrasts with the *compositional* approach typically taken in denotational descriptions, such as in [Schmidt 1986], and the *structural* approach taken in modern operational semantics, as in [Hennessy 1990]. Our model differs from these approaches since, for both compositional and structural approaches, the definition is structured around the (abstract) syntax of the programming language being defined. This makes sense for high-level languages which express control using nested syntactic structures (such as if statements, while-loops etc.), but is not so appropriate to Forth, where a program is simply a sequential list of words, with no real nesting.

We take a different approach, making the processes of the Forth system central to our model, and viewing the program text as a stream of events which affect these processes. In this, our semantics bears a similarity to previous specifications of aspects of Forth, such as [Knaggs 1993] and particularly [Stoddart 1996]. However, these specifications concentrated on the execution of Forth programs, whereas our specification also incorporates the actions of the Forth compiler. An alternative approach [Pöial 1993] employs Hoare-style axioms to a fragment of Forth, but appears to treat compile-time words as primitives in the language. More recently [Pöial 2003] presents a type system for inferring stack safety for similar fragments of Forth, but also does not deal with compile-time semantics.

The modelling formalism that we have chosen to use here is the π-calculus [Milner 1999, Sangiorgi and Walker 2001]. The π-calculus provides a primitive set of operations for describing the interactions between communicating processes, as well as allowing for the movement of communication channels between these processes. This formalism is not typically used to specify the formal semantics of programming languages ([Röckl and Sangiorgo 1999] is an exception, but even this concentrates on concurrency aspects of the language).

However, we believe that the π-calculus is particularly suited to providing an operational model of the web of interactions between various components of the Forth system. To model the Forth system it is necessary to allow for a number of interacting processes which may include either compile-time or run-time behaviour, as the system switches from compiler to interpreter mode. In addition, the use of immediate and postponed words within word definitions further enmeshes these processes, making their presentation using standard structural approaches quite difficult. Also, the basic purpose of the Forth compiler, the definition of new words, finds a natural model in the π-calculus primitives for the creation and movement of names between processes, and the necessary reconfiguration of communications between these processes.

We will not give a full presentation of the π-calculus here; we seek only to introduce the notation used in the rest of this paper. The two main *events* that can occur are:

- $c(n)$, denoting the receipt of some message, hereby named n, along a channel c, and
- $\overline{c}\langle n\rangle$, denoting the sending of some existing name n out along the channel c

In the π-calculus channels are first-order objects; that is, channels can be sent and received along other channels. The silent event, τ denotes an internal action, hidden to other processes.

For our purposes, *processes* can then be described as:

- $e.P$, where e is an event and P is a process - the process P here is guarded by the event e

- $P_1 + P_2$ denoting nondeterministic choice between processes P_1 and P_2
- $P_1 \mid P_2$, which denotes the processes P_1 and P_2 being run in parallel
- new a (P) introducing (and binding) the new name a in process P
- $P_1; P_2$, which denotes the sequential composition of processes P_1 and P_2[1]

Two processes running in parallel may use guards to synchronise; the basic *reaction rule* formalises the synchronisation in a manner similar to β-reduction in the lambda calculus:

$$(x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \quad \rightarrow \quad \{z/y\}P \mid Q$$

2.3 Notation

To increase the readability of our specification, we use some notational conveniences not primitive to the π-calculus:

- We use macro-like definitions to give names to processes; thus $P(n) \stackrel{\text{def}}{=} Q$ defines the macro P indexed by n, defined to be the same as process Q after suitable substitution for parameter n.
- Tuples are represented using square brackets; for example: $[x, y]$ is the pair consisting of x and y.
- We use $l_1{}^\wedge l_2$ to represent the concatenation of lists l_1 and l_2, and overload this notation to also apply to list elements. We use the constant Nil to represent the empty list, and the functions hd and tl to represent the usual list head and tail operations.
- If both s_1 and s_2 are strings we will use $s_1{}^\wedge s_2$ to represent their (string) concatenation.

All of the specifications presented in the following sections have been type-checked and tested using the Pict system [Pierce and Turner 1997], a programming language based closely on the π-calculus. The specification was translated almost directly into Pict, which helped to check the consistency of the model, and, as an executable language, allowed us to simulate the actions of the compiler as it dealt with various configurations of Forth source code. The full Pict source code is presented in [Power and Sinclair 2001].

3 Overview of the Model

The Forth programming language is characterised by the fact that it is not only a stack-based language, but also a semi-compiled language. A Forth program

[1] If we assume that every process performs the action $\overline{done}\langle\rangle$ as its last action we can define sequential composition as a special case of parallel composition.
$P; Q \stackrel{\text{def}}{=}$ new $start$ $(\{start/done\}P \mid start.Q)$

```
ForthProgram =
   { WordDefinition }, Ident
 ;
WordDefinition =              (* Define an Ident as a sequence of Commands *)
   ":", Ident, { Command }, ";"                          (* Ordinary word *)
 | ":", Ident, { Command }, "; immediate"               (* Immediate word *)
 ;
Command =
   DataCommand
 | CFSCommand
 | ImmediateCommand
 | "postpone", ImmediateCommand
 | "postpone", Ident                   (* Ident must be an immediate word *)
 | Ident                                 (* Call/Execute the word Ident *)
 | "recurse"                         (* Recursive call to the current word *)
 | "exit"                            (* Return control to the calling word *)
 ;
DataCommand =                     (* Commands that manipulate the data stack *)
   "drop" | "dup" | "swap" | "tuck"         (* Stack manipulation instruction *)
 | "+" | "-" | "*" | "/"                    (* Standard arithmetic operations *)
 | "=" | "<" | ">"                          (* Standard comparison operations *)
 | Number                                    (* Push a number on to the stack *)
 ;
CFSCommand =              (* Commands that manipulate the control flow stack *)
   "cs-roll"                                     (* Reorder the control stack *)
 | "cs-pick"                             (* Copy an item on the control stack *)
 ;
ImmediateCommand =        (* Commands that are executed during compilation *)
   "if"                     (* Mark the origin of a forward conditional branch *)
 | "ahead"              (* Mark the origin of a forward unconditional branch. *)
 | "begin"                                      (* Mark a backward destination *)
 | "then"                                          (* Resolve a forward branch *)
 | "again"                       (* Resolve a backward unconditional branch *)
 | "until"                          (* Resolve a backward conditional branch *)
 ;
```

Figure 1: An EBNF definition of the abstracted Forth syntax used in this paper. *Here we assume* Ident *represents an identifier, and* Number *an integer literal.*

---

consists of a series of words, which can be either Forth standard words or words defined by the user. These words are processed sequentially by the *interpreter*. Each word has an entry in the *dictionary* and the compiled code associated with the word read by the interpreter is retrieved from the dictionary and executed. The effect of each word is to modify the state of one of the Forth stacks.

The Forth language has three stacks. The *data stack* is used to manipulate values; for this paper, we assume all these values are integers. The *return stack* is used to restore the state after a Forth word has been called. The *control-flow stack* is used to implement control flow management. Forth provides a set of standard words that provide a more structured approach to control flow management than the basic low-level branches and labels implemented by many assembly languages.

In this paper we are interested in Forth's approach in providing structured control flow primitives, as we believe that this has a fundamental impact on verification and optimisation, as well as a possible influence on the design of future intermediate representations. We will abstract the syntax of a Forth program to the primitives shown in Figure 1.

We define a Forth program as consisting of a series of word definitions followed by an invocation of one of those words. When a ":" is encountered, a new word is being defined, and the Forth system switches from *interpreter mode* into *compiler mode* until the subsequent ";" or "; IMMEDIATE" is encountered. Each new Forth word is defined in terms of existing Forth words, which are either immediate words or non-immediate words. Immediate words are executed immediately by the compiler whereas non-immediate words have a call to their compiled code appended to the compiled code of the Forth word being defined. It is only when the non-immediate word is subsequently executed by the interpreter that the behaviour defined by the non-immediate word is executed. An immediate word may be preceded by POSTPONE, causing it to be treated as a non-immediate word.

As well as primitives for defining new words, we include the six standard immediate control words IF, THEN, BEGIN, AGAIN, UNTIL and AHEAD. These control the generation of branch and labelled instructions by maintaining all labels in the program on the control flow stack. Each of these instructions corresponds to either generating a new label or using an existing label from the control flow stack. The primitives CS-ROLL and CS-PICK are used to change the ordering on the control flow stack without generating any new labels.

We note that the presence of immediate words, with behaviour that is executed at compile-time, considerably complicates the static analysis of Forth programs. Indeed, without the ability to execute such words during the analysis, static analyses such as stack safety checks cannot be fully implemented on Forth code. While immediate words are specific to Forth, echoes of this difficulty can be found in the analysis of C++ programs, where techniques such as template meta-programming [Veldhuizen 1995] can build considerably functionality into the compile-time processing of a program.

Figure 2 gives an overview of our specification of the Forth programming system. It shows the processes that model each component of the Forth system and the channels that carry information between the processes, and should thus be used as a reference when reading the π-calculus specification in Sections 4 and 5. Section 6 provides examples of the interactions between these processes.

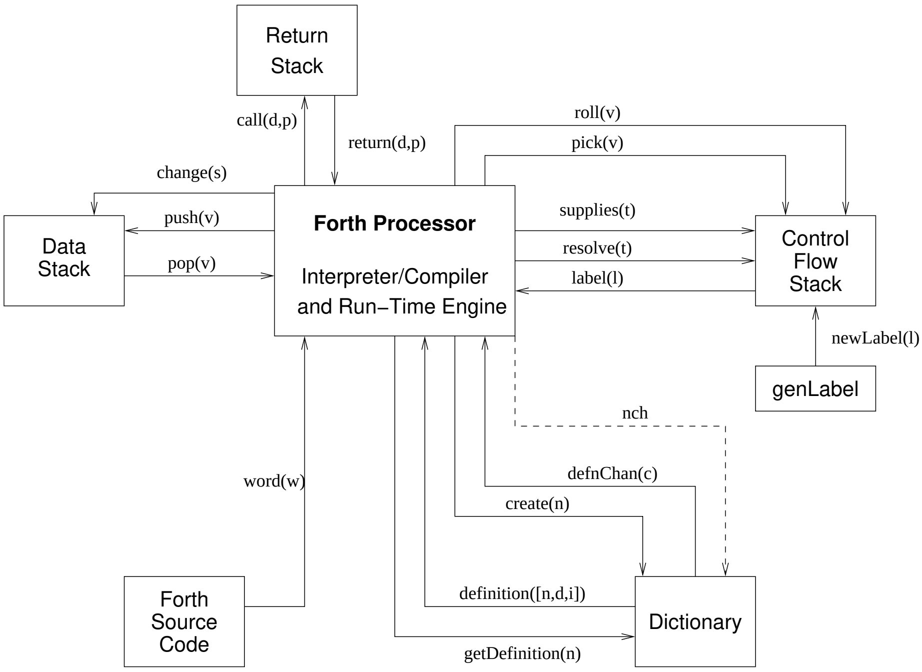

Figure 2: Overview of the specification of the Forth system. *The processes and events depicted here are described by the specifications in Sections 4 and 5.*

$$
\begin{aligned}
&FORTH(prog, dict) \stackrel{\text{def}}{=} \\
&\quad INTERP \mid SOURCE(prog) \mid DICTIONARY(dict) \\
&\quad \mid DATA(Nil) \mid CFS(Nil) \mid RETURN(Nil) \mid genLabel(1)
\end{aligned}
$$

Figure 3: Specification of the main processes of the Forth system. *This is the starting point for the processing of some Forth program* prog*, based on an initial dictionary* dict *of pre-defined words.*

---

4 Processing a Forth Program

Figure 3 is our top level specification of the Forth programming system. The system is the parallel composition of the interpreter process *INTERP*, supplied with Forth source code from *SOURCE*, using the dictionary process *DICTIONARY*, and manipulating the three stacks - the data stack, *DATA*, the control-flow stack, *CFS*, and the return stack, *RETURN*.

In the remainder of this section we describe those processes within the Forth system that interact with the Forth source code, specifically the interpreter,

$$
\begin{aligned}
&\mathit{INTERP} \stackrel{\mathtt{def}}{=} \\
&\quad \mathit{word}(w). \\
&\quad \text{if } w = \text{“:”} \text{ then } \mathit{word}(n).\overline{\mathit{create}}\langle n\rangle.\mathit{defnChan}(\mathit{nch}).\mathit{COMPILE}(n, \mathit{nch}) \\
&\qquad\qquad\quad\ \text{else } \ \mathit{EXECUTE}(w).\mathit{INTERP}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SOURCE}(\mathit{prog}) \stackrel{\mathtt{def}}{=} \\
&\quad \overline{\mathit{word}}\langle \mathit{hd\ prog}\rangle.\mathit{SOURCE}(\mathit{tl\ prog})
\end{aligned}
$$

Figure 4: Specification of the main Forth interpreter loop and the source code process which feeds it.

---

dictionary and compiler. In Section 5 we describe the remaining processes that deal with compile-time semantics and executing the code.

4.1 The main interpreter loop

Figure 4 elaborates the definition of the interpreter process, showing its two main sub-processes, the Forth compiler process, *COMPILE*, and the run-time engine process, *EXECUTE*.

When the interpreter reads a word definition, which starts “: n”, it creates an entry in the dictionary for the word n and invokes the Forth compiler with the supplied channel *nch*. The compiler translates the word, feeds its definition to the dictionary, and eventually returns control to the interpreter. If the interpreter encounters a word that does not begin a definition then it is executed directly through the *EXECUTE* process.

Both the compilation and interpretation processes in Forth are driven by the source code of the program being processed, in the form of a stream of Forth words. This is modelled in Figure 4 as a simple list of words, represented by the *SOURCE* process, and these are supplied one at a time to the interpreter along the *word* channel.

4.2 The Forth Dictionary

The Forth dictionary is used to hold word definitions, and is defined in Figure 5 as the *DICTIONARY* process. The two primary operations of the dictionary are to add a new word to the dictionary or report on an existing word.

When adding a new word to the dictionary a new channel *nch* is created specifically for that word at the request of the interpreter. This channel can then be used by the the compiler to supply words from the definition to the dictionary. When the definition is completed then the word n is marked as either “ORD” or

$$
\begin{aligned}
&DICTIONARY(w) \stackrel{\mathtt{def}}{=} \\
&\quad create(n). \\
&\qquad \text{new } nch\ (DICTADD(nch, [n, Nil, \text{"}\textsc{ord}\text{"}], w) \mid \overline{defnChan}\langle nch\rangle) \\
&\ + getDefinition(n).\overline{definition}\langle(find\ n\ w)\rangle.DICTIONARY(w)
\end{aligned}
$$

$$
\begin{aligned}
&DICTADD(nch, [n, d, i], w) \stackrel{\mathtt{def}}{=} \\
&\quad nch(s). \\
&\qquad\qquad \text{if } s = done \text{ then } DICTIONARY([n, d, i]\hat{}\,w) \\
&\quad \text{else if } s = doneimm \text{ then } DICTIONARY([n, d, \text{"}\textsc{imm}\text{"}]\hat{}\,w) \\
&\qquad\qquad\qquad\qquad \text{else } \ DICTADD(nch, [n, d\hat{}\,s, i], w)
\end{aligned}
$$

Figure 5: Specification of the Forth dictionary process which allows word definitions to be created and accessed.

"IMM", depending on whether the definition is an ordinary word terminated with ";" or an immediate word terminated with "; IMMEDIATE", and the channel is released.

Reporting on a word n involves searching the dictionary and returning its definition d and its immediacy status i. Here the function *find* returns a dictionary entry for a given word n, or *Nil* if the word is not defined.

4.3 A Simple Assembly Language

In order to describe the operation of the Forth compiler we must consider its output - the compiled code which is stored in the dictionary. To describe this generated code we use a simple assembly language, which we will refer to as SAL consisting mainly of labels and jumps, whose syntax is described in Figure 6. For simplicity we use the Forth data stack and control-flow stack manipulation instructions defined in Figure 1 directly in this assembly language.

We note the similarity here with the $\mathrm{JVM}_{\mathcal{I}}$ language that has been specified to model the imperative core of Java [Börger and Schulte 1998]. Our language *SAL* is quite similar to $\mathrm{JVM}_{\mathcal{I}}$, except that it does not require `"load"` and `"store"` instructions, since core Forth does not maintain a separate local variable array.

4.4 The Forth Compiler

Figure 7 specifies the Compiler process. The compiler reads successive source code words until it reads either ";" or "; IMMEDIATE", relays the appropriate signal to the dictionary, and then returns to the top-level interpreter process.

```
SimpleAssemblyLanguage =
   {AssemblyInstruction}        (* A sequence of (possibly labelled) instructions *)
;
AssemblyInstruction =
   "call", Ident                (* Call another word in the Dictionary *)
 | "return"                     (* Return control to caller *)
 | Ident, ":"                   (* A label *)
 | "goto", Ident                (* Unconditional branch to labelled instruction *)
 | "ifzero", Ident              (* Conditional branch to labelled instruction *)
 | DataCommand                  (* As for Forth (see Figure 1) *)
 | CFSCommand                   (* As for Forth (see Figure 1) *)
;
```

Figure 6: An EBNF definition of the syntax of SAL, a Simple Assembly Language.

$$
\begin{array}{l}
\mathit{COMPILE}(n, nch) \stackrel{\mathtt{def}}{=} \\
\quad \mathit{word}(w). \\
\qquad \begin{array}{rl}
\text{if } \mathit{isDataCommand}(w) & \text{then } \overline{nch}\langle w\rangle.\mathit{COMPILE}(n, nch) \\
\text{else if } \mathit{isCFSCommand}(w) & \text{then } \overline{nch}\langle w\rangle.\mathit{COMPILE}(n, nch) \\
\text{else case } (w) \text{ of} & \\
\text{“;”} & \text{then } \overline{nch}\langle \mathit{done}\rangle.\mathit{INTERP} \\
\text{“; IMMEDIATE”} & \text{then } \overline{nch}\langle \mathit{doneimm}\rangle.\mathit{INTERP} \\
\text{“POSTPONE”} & \text{then } \mathit{word}(m).\overline{nch}\langle m\rangle.\mathit{COMPILE}(n, nch) \\
\text{“RECURSE”} & \text{then } \overline{nch}\langle \texttt{"call "}\hat{\ } n\rangle.\mathit{COMPILE}(n, nch) \\
\text{“EXIT”} & \text{then } \overline{nch}\langle \texttt{"return"}\rangle.\mathit{COMPILE}(n, nch) \\
 & \text{else } \overline{getDefinition}\langle w\rangle.definition(defn). \\
\text{if } defn = [w, d, \text{“ORD”}] & \text{then } \overline{nch}\langle \texttt{"call "}\hat{\ } w\rangle.\mathit{COMPILE}(n, nch) \\
\text{else if } defn = [w, d, \text{“IMM”}] & \text{then } \mathit{IMMEDIATE}(n, nch, d) \\
\text{else if } defn = Nil & \text{then } \mathit{IMMEDIATE}(n, nch, [w])
\end{array}
\end{array}
$$

Figure 7: Specification of the main actions of the Forth interpreter and compiler. *The immediate words are dealt with by the* IMMEDIATE *process, described later.*

If the compiler reads a data stack or control-flow stack operation it relays it directly to the compiled code. Here we assume boolean-valued functions *isDataCommand* and *isCFSCommand* to syntactically test if a word is a data stack or control-flow stack operation respectively. If, during compilation, the compiler reads the word POSTPONE it then reads the next word m from the source code and sends that word along channel *nch* to the dictionary.

If the compiler reads any other word it checks for a dictionary entry for that word. If w is a non-immediate word a subroutine call (represented as `"call"`) to the compiled code for w is generated and added to the n's entry in the dictionary.

$$
\begin{aligned}
&CFS([l_0,t_0]\hat{}[l_1,t_1]\hat{}\cdots\hat{}[l_n,t_n]) \stackrel{\text{def}}{=} \\
&\qquad resolve(t_0).\overline{label}\langle l_0\rangle.CFS([l_1,t_1]\hat{}\cdots\hat{}[l_n,t_n]) \\
&\quad + supplies(t').newLabel(l').\overline{label}\langle l'\rangle.CFS([l',t']\hat{}[l_0,t_0]\hat{}[l_1,t_1]\hat{}\cdots\hat{}[l_n,t_n]) \\
&\quad + pick(k).CFS([l_k,t_k]\hat{}[l_0,t_0]\hat{}[l_1,t_1]\hat{}\cdots\hat{}[l_n,t_n]) \\
&\quad + roll(k).CFS([l_k,t_k]\hat{}[l_0,t_0]\hat{}[l_1,t_1]\hat{}\cdots[l_{k-1},t_{k-1}]\hat{}[l_{k+1},t_{k+1}]\hat{}\cdots\hat{}[l_n,t_n])
\end{aligned}
$$

$$
\begin{aligned}
&genLabel(n) \stackrel{\text{def}}{=} \\
&\quad \overline{newLabel}\langle n\rangle.genLabel(n+1)
\end{aligned}
$$

Figure 8: Specification of the Forth control-flow stack (*CFS*). *This stack is used for label management, and maintained by the compiler as it processes immediate words.*

If w is an immediate word, or if its definition is not found, the *IMMEDIATE* process, described below in Section 5, takes over to execute the immediate semantics of the word.

5 Compile-Time and Run-Time Execution

In this section we turn to the main focus of our paper, the formal specification of Forth control words. In particular we specify the compile-time activities of the Forth processor, mainly centred around label management via the control-flow stack, as well as code generation and execution.

The execution of immediate words will be described after the control-flow stack is specified since it relies on the control-flow stack to generate the structured control flow.

5.1 The Control-Flow Stack

The control-flow stack stores [*label*,*type*] pairs that are used by the control flow words IF, THEN, BEGIN, AGAIN, UNTIL and AHEAD, as described in Figure 1. The valid label types are *orig* and *dest*. Pairs are added to the control-flow stack via the *supplies* channel, which also causes the creation of a new label along *newLabel*. A pair is removed from the control-flow stack when the value on the *resolve* channel matched the *type* of the pair on top of the control-flow stack. In each case, the *label* value of the pair is transmitted by the control-flow stack to the compiler via the *label* channel. The *pick* and *roll* channels are used to modify the control-flow stack in response to CS-PICK and CS-ROLL instructions.

$$
\begin{array}{l}
\mathit{IMMEDIATE}(n, nch, ws) \stackrel{\mathtt{def}}{=} \\
\quad \text{if } ws = Nil \text{ then } \mathit{COMPILE}(n, nch) \\
\qquad\qquad\qquad \text{else } \mathit{IMM}(nch, (hd\ ws)).\mathit{IMMEDIATE}(n, nch, (tl\ ws))
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{IMM}(nch, w) \stackrel{\mathtt{def}}{=}} \\
\multicolumn{2}{l}{\quad \text{case } (w) \text{ of}} \\
\text{“IF”} & \text{then } \overline{supplies}\langle orig\rangle.label(l).\overline{nch}\langle \texttt{"ifzero "}\hat{}\,l\rangle \\
\text{“THEN”} & \text{then } \overline{resolve}\langle orig\rangle.label(l).\overline{nch}\langle l\hat{}\,\texttt{" :"}\rangle \\
\text{“BEGIN”} & \text{then } \overline{supplies}\langle dest\rangle.label(l).\overline{nch}\langle l\hat{}\,\texttt{" :"}\rangle \\
\text{“AGAIN”} & \text{then } \overline{resolve}\langle dest\rangle.label(l).\overline{nch}\langle \texttt{"goto "}\hat{}\,l\rangle \\
\text{“UNTIL”} & \text{then } \overline{resolve}\langle dest\rangle.label(l).\overline{nch}\langle \texttt{"ifzero "}\hat{}\,l\rangle \\
\text{“AHEAD”} & \text{then } \overline{supplies}\langle orig\rangle.label(l).\overline{nch}\langle \texttt{"goto "}\hat{}\,l\rangle \\
 & \text{else } \mathit{STACKEXEC}(w)
\end{array}
$$

Figure 9: Specification of the execution of Forth immediate words. *This specification should be considered in conjunction with that of the control-flow stack, since its action mainly involve accessing the labels on that stack.*

5.2 Executing Immediate Words at Compile Time

Figure 9 specifies the execution of immediate words at compile-time. Here, *IMMEDIATE* uses the sub-process *IMM* to process each of the words one at a time. When the *IMM* process receives the immediate Forth words IF, THEN, BEGIN, AGAIN, UNTIL or AHEAD it performs the corresponding actions on the control-flow stack, and generates either a label, conditional jump or unconditional jump as appropriate.

For example, in the case of the conditional forward branch (IF) a request for a label of type *orig* is sent to the control-flow stack and the label l is then received along *label*. The conditional branch `"ifzero l"` is sent along the code channel. A subsequent THEN word issues a $\overline{resolve}\langle orig\rangle$ to the control-flow stack and receives a label l. The label `"l:"` will then be appended to the compiled code.

The process *STACKEXEC* carries out an stack-based operation on the data stack or control flow stack as appropriate, and is defined in Figure 10 below.

5.3 The Data Stack

Figure 10 specifies the data stack process, where we model this stack as a list of integer values ($v_0\hat{}v_1\hat{}\cdots\hat{}v_n$). Values can be added to the front of the list via the *push* channel or removed from the front of the list in response to a *pop* signal. The *change* channel allows us to send those commands that change the stack without any other input or output. As well as some standard Forth stack manipulation

$$
\begin{array}{l}
DATA(v_0 {}^\wedge v_1 {}^\wedge \cdots {}^\wedge v_n) \stackrel{\texttt{def}}{=} \\
\qquad push(v).DATA(v {}^\wedge v_0 {}^\wedge v_1 {}^\wedge \cdots {}^\wedge v_n) \\
\quad + \overline{pop}\langle v_0 \rangle.DATA(v_1 {}^\wedge \cdots {}^\wedge v_n) \\
\quad + change(w).\ \text{case } (w) \text{ of} \\
\qquad \text{“DROP” then } DATA(v_1 {}^\wedge \cdots {}^\wedge v_n) \\
\qquad \text{“DUP” then } DATA(v_0 {}^\wedge v_0 {}^\wedge v_1 {}^\wedge \cdots {}^\wedge v_n) \\
\qquad \text{“SWAP” then } DATA(v_1 {}^\wedge v_0 {}^\wedge \cdots {}^\wedge v_n) \\
\qquad \text{“TUCK” then } DATA(v_0 {}^\wedge v_1 {}^\wedge v_0 {}^\wedge \cdots {}^\wedge v_n) \\
\qquad\qquad op \text{ then } DATA((f_{op}\ v_1\ v_0) {}^\wedge \cdots {}^\wedge v_n) \\
\qquad\qquad\qquad where\ op \in \{\text{“+”}, \text{“-”}, \text{“*”}, \text{“/”}, \text{“=”}, \text{“<”}, \text{“>”}\} \\
\qquad\qquad\qquad and\ f_{op} \text{ is the corresponding integer operation}
\end{array}
$$

$$
\begin{array}{l}
STACKEXEC(w) \stackrel{\texttt{def}}{=} \\
\qquad\qquad \text{if } (w = \text{“CS-PICK”}) \text{ then } pop(v).\overline{pick}\langle v \rangle \\
\qquad \text{else if } (w = \text{“CS-ROLL”}) \text{ then } pop(v).\overline{roll}\langle v \rangle \\
\qquad\quad \text{else if } isNumber(w) \text{ then } \overline{push}\langle w \rangle \\
\quad \text{else if } isDataCommand(w) \text{ then } \overline{change}\langle w \rangle
\end{array}
$$

Figure 10: Specification of *DATA*,the Forth data stack, and *STACKEXEC*, a process that dispatches instructions to the control-flow and data stack as appropriate.

---

commands we include integer arithmetic and comparison operations. Each of the operations received along the *change* channel corresponds directly to a operation that can be used in the Forth source code.

The *STACKEXEC* process, also defined in Figure 10, handles requests from either the compiler or run-time engine to access either the control-flow stack or the data stack. We note that both of the control-flow stack instructions “CS-PICK” and “CS-ROLL” first retrieve an integer from the data stack before performing the appropriate operation on the control-flow stack.

5.4 The Run-Time Engine

The final step in our definition is to give a run-time semantics for the compiled word definitions. Through the specifications given above, these Forth definitions have been compiled into the simple assembly language SAL of Figure 6. The semantics of SAL are given in Figure 11.

The process *EXECUTE* was called by the interpreter in Figure 7, and its purpose is to execute a word whose definitions in SAL has been retrieved from the dictionary. It does this by calling the sub-process *EXEC* which effectively maintains a program counter as we step through the code.

$$
\begin{array}{l}
\mathit{EXECUTE}(w) \stackrel{\mathtt{def}}{=} \\
\quad \mathit{EXEC}([\texttt{"call "}\hat{~}w, \texttt{"eof"}], 0)
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{EXEC}(d,p) \stackrel{\mathtt{def}}{=}} \\
\multicolumn{2}{l}{\quad \text{case } (at(d,p)) \text{ of}} \\
\texttt{"eof"} \text{ then} & \tau \\
(\texttt{"call "}\hat{~}w_1) \text{ then} & \overline{call}\langle[d,p+1]\rangle. \\
& \overline{getDefinition}\langle w_1\rangle.definition([w_1, d_1, \text{“ORD”}]). \\
& \mathit{EXEC}(d_1, 0) \\
\texttt{"return"} \text{ then} & return([d_1,p_1]).\mathit{EXEC}(d_1,p_1) \\
(l\hat{~}\texttt{" :"}) \text{ then} & \tau.\mathit{EXEC}(d,p+1) \\
(\texttt{"goto "}\hat{~}l) \text{ then} & \mathit{EXEC}(d, locate(d,l)) \\
(\texttt{"ifzero "}\hat{~}l) \text{ then} & pop(v). \\
& \text{if } (v=0) \text{ then } \mathit{EXEC}(d, locate(d,l)) \\
& \text{else } \mathit{EXEC}(d,p+1) \\
\text{else} & \mathit{STACKEXEC}(at(d,p)).\mathit{EXEC}(d,p+1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{RETURN}(w) \stackrel{\mathtt{def}}{=} \\
\quad call([d,p]).\mathit{RETURN}([d,p]\hat{~}w) \\
\quad + \overline{return}\langle(hd\ w)\rangle.\mathit{RETURN}(tl\ w)
\end{array}
$$

Figure 11: Specification of *EXECUTE*, the run-time execution engine, and *RETURN*, the return stack.

The *EXEC* process is parameterised by a list of assembly SAL instructions, being the definition of a Forth word, and a location within that list. As well as a special case for `"eof"` denoting completion, the *EXEC* process has one case for each of the five kinds of assembly instructions, along with two cases handling instructions for the data stack and control-flow stack.

The `"call"` and `"return"` instructions push and pop the program counter to the return stack, while the conditional and unconditional jumps simply modify the current program counter. To aid us in this we assume the existence of a function $at(d, p)$ returning the instruction at position p in list d, and a function $locate(d, l)$ returning the location of label l in the list d.

The return stack is simply a stack of program counters. Its two channels, *call* and *return* are just appropriate versions of push and pop operations. The definition of the return stack is given as the process *RETURN* in Figure 11

```
: while                   : repeat                  : fact_w
  postpone if               postpone again            begin
  1 cs-roll                 postpone then                 dup 0 >
; immediate               ; immediate                 while
                                                          tuck * swap 1 -
                                                      repeat
                                                    ;
```

Figure 12: Standard definitions of the Forth control words WHILE and REPEAT, along with a definition of FACT_W, which uses them to calculate the factorial of a number.

6 Some Examples of Compiling Forth Control Words

In this section we provide an example of the use of the semantics given in previous sections. In particular, we show how Forth's IF/THEN, WHILE and RECURSE constructs are handled by our semantics, and we prove the equivalence of two definitions of a factorial program, one using the WHILE construct, the other using recursion. The approach here should be contrasted with high-level languages with built-in nested control structures, as well as with assembly and virtual machine languages that must rely on unstructured test and branch instructions.

We have simulated the operation of the Forth compiler over these and other examples through a translation of the π-calculus specification into the Pict programming language. Because of the close relationship between Pict and the π-calculus there is a strong correspondence between the specifications presented in previous sections and the Pict code, apart from some minor administrative details. The Pict simulation takes a list of Forth word definitions corresponding to the syntax in Figure 1, and translates them into the simple assembly language, SAL, of Figure 6, executing any compile-time semantics in the process. The Pict simulation then uses the basic run-time semantics for SAL to run the program, producing an appropriate result on the data stack.

6.1 An Example of Compile-Time Semantics

In this subsection we demonstrate the working of the *COMPILE* and *IMMEDIATE* processes and their interaction with the control-flow stack. The example we use is a simple implementation of the factorial function using a loop. To define an equivalent of the `while` loop in Forth, we follow the ISO standard and define two extra control words, WHILE and REPEAT, and use these in our definition of the factorial function, FACT_W. The definitions are given in Figure 12.

The compiler will first process the definitions of the two immediate words, storing a definition of WHILE as "IF 1 CS-ROLL" and a definition of REPEAT as "AGAIN THEN". Since all immediate words in these definitions are preceded by POSTPONE, no compile-time evaluation is done here.

Process	Word	Control Flow Stack	Output
INTERP	: FACT_W		"fact_w = "
IMMEDIATE	BEGIN	[L_1,dest]	"L_1 :"
COMPILE	DUP 0 >	[L_1,dest]	"dup 0 >"
COMPILE	WHILE	[L_1,dest]	
IMMEDIATE	IF	[L_1,dest] [L_2,orig]	"ifzero L_2"
IMMEDIATE	1 CSROLL	[L_2,orig] [L_1,dest]	
COMPILE	TUCK * SWAP 1 -	[L_2,orig] [L_1,dest]	"tuck * swap 1 -"
COMPILE	REPEAT	[L_2,orig] [L_1,dest]	
IMMEDIATE	AGAIN	[L_2,orig]	"goto L_1"
IMMEDIATE	THEN		"L_2 :"

Figure 13: Compiling the definition of program `fact_w`. *The table shows the process, the current Forth word , the labels on the control-flow stack, and the generated* SAL *code.*

However, when processing the definition of FACT_W, three compile-time words are used: the built-in word BEGIN, and the two immediate words that we have just defined. Figure 13 provides an outline trace of the actions of the processes in the specification when processing FACT_W. As can be seen, the immediate words manipulate the labels on the control-flow stack to provide the correct SAL output

As mentioned earlier, Forth programmers are not restricted to a predefined set of control structures, but are free to devise new combinations of the control words to construct more exotic control patterns. Appendix A.3 of the ISO standard gives some examples of Forth versions of common and not-so-common control patterns.

6.2 An equivalence proof

As well as providing a formal definition of the compilation process, our specification also provides the basis for proving equivalence between programs. In this subsection we demonstrate the use of the run-time semantics in proving equivalence between the while-loop version of factorial given in the previous subsection, and a recursive version.

We can define a recursive version of factorial using the immediate words IF and THEN which generate and resolve a single label. A recursive definition of a factorial word, along with its corresponding output in SAL, is shown in Figure 14.

Let us refer to the Forth program consisting of the definitions of FACT_W and FACT_R from Figures 13 and 14 as as FACT_FS. Each version of the factorial function needs a counter on the stack to start with, and this can be disposed of

Process	Word	Control Flow Stack	Output
INTERP	: FACT_R		`"fact_r = "`
COMPILE	DUP 0 >	[L_2,orig]	`"dup 0 >"`
IMMEDIATE	IF	[L_2,orig]	`"ifzero L_2"`
COMPILE	TUCK * SWAP 1 -	[L_2,orig]	`"tuck * swap 1 -"`
COMPILE	RECURSE	[L_2,orig]	`"call fact_r"`
IMMEDIATE	THEN		`"L_2 :"`

Figure 14: Compiling the definition of program `fact_r`. *We have numbered the (sole) label here as* `"L_2"` *to facilitate comparison with the* FACT_W

after the calculation; thus for convenience we define the syntactic macro:

$$test(f, n) = [1, n, f, \text{DROP}]$$

We would like to show that for any natural number n, the while-loop and recursive versions of the factorial program are equivalent from the point of view of the data stack.

$$\begin{aligned} &\mathit{FORTH}(\text{FACT\_FS}\hat{\ }test(\text{FACT\_W}, n), Nil) \\ &\quad \approx \mathit{FORTH}(\text{FACT\_FS}\hat{\ }test(\text{FACT\_R}, n), Nil) \end{aligned}$$

That is, ignoring the effect of `"call"` and `"return"` instructions on the return stack, the two programs are functionally equivalent. We do not have space here to follow the full set of transitions through, but we present an outline of the proof below, highlighting the impact on the processes *DATA* and *EXEC*. [2]

6.2.1 Proof Outline

The proof is by induction on the value on top of the data stack before the execution of FACT_W or FACT_R, which we refer to as n.

Case $n = 0$ (inductive base)

Here, with 0 on the data stack, the rules of Figure 10 give rise to the set of reductions for `fact_w`

$$\begin{aligned} &\mathit{DATA}(0\hat{\ }\cdots) \mid \mathit{EXEC}(\texttt{"L\_1:"}, 0) \\ &\stackrel{\tau}{\Longrightarrow} \mathit{DATA}(0\hat{\ }\cdots) \mid \mathit{EXEC}(\texttt{"dup"}, 1) \\ &\stackrel{change(dup)}{\Longrightarrow} \mathit{DATA}(0\hat{\ }0\hat{\ }\cdots) \mid \mathit{EXEC}(\texttt{"0"}, 2) \\ &\stackrel{push(0)}{\Longrightarrow} \mathit{DATA}(0\hat{\ }0\hat{\ }0\hat{\ }\cdots) \mid \mathit{EXEC}(\texttt{">"}, 3) \\ &\stackrel{change(>)}{\Longrightarrow} \mathit{DATA}(0\hat{\ }0\hat{\ }\cdots) \mid \mathit{EXEC}(\texttt{"ifzero L\_2"}, 4) \end{aligned}$$

[2] The *EXEC* process is parameterised by the current word definition and a position in that definition. To simplify the proof presentation, we show only the “current” word in the definition as the first parameter of *EXEC*.

The transitions for `fact_r` are almost identical for these instructions, except for the first label. After executing the comparison instructions the counter is just before the conditional jump, and from Figure 11 we can deduce that we execute the jump instruction, leading to:

$$\begin{aligned}&\mathit{DATA}(0\hat{}0\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"ifzero L\_2"}, 4)\\ &\quad\stackrel{*}{\Longrightarrow}\mathit{DATA}(0\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"L\_2 :"}, 11)\end{aligned}$$

Clearly, this is equivalent for both `fact_w` and `fact_r`, since in each case `"L_2 :"` is at the end of the word definition.

Case $n > 0$ (inductive step)

By the opposite argument to that given in the base case, we can show that when n is on the top of the stack:

$$\begin{aligned}&\mathit{DATA}(n\hat{}n\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"ifzero L\_2"}, 4)\\ &\quad\stackrel{*}{\Longrightarrow}\mathit{DATA}(n\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"tuck"}, 5)\end{aligned}$$

From Figure 11 and then Figure 10 we can show that from this state, with n on top of the data stack, and v as the answer just underneath it, we get the following series of transitions:

$$\begin{aligned}&\mathit{DATA}(n\hat{}v\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"tuck"}, 5)\\ &\quad\stackrel{change(tuck)}{\Longrightarrow}\mathit{DATA}(n\hat{}v\hat{}n\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"*"}, 6)\\ &\quad\stackrel{change(*)}{\Longrightarrow}\mathit{DATA}((v*n)\hat{}n\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"swap"}, 7)\\ &\quad\stackrel{change(swap)}{\Longrightarrow}\mathit{DATA}(n\hat{}(v*n)\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"1"}.8)\\ &\quad\stackrel{push(1)}{\Longrightarrow}\mathit{DATA}(1\hat{}n\hat{}(v*n)\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"-"}, 9)\\ &\quad\stackrel{change(-)}{\Longrightarrow}\mathit{DATA}((n-1)\hat{}(v*n)\hat{}\cdots) \mid \mathit{EXEC}(\texttt{"goto L\_1"}, 10)\end{aligned}$$

The important point here is that the execution of these instructions has decremented the value on top of the stack by 1, which gives us the link to the inductive hypothesis.

The jump instruction in `fact_w` returns control to the start of the loop body, so we have that:

$$\begin{aligned}&\mathit{DATA}(n\hat{}\cdots) \mid \mathit{EXEC}([\texttt{"L\_1:"}, \cdots, \texttt{"goto L\_1"}], 0)\\ &\quad\stackrel{*}{\Longrightarrow}\mathit{DATA}((n-1)\hat{}\cdots) \mid \mathit{EXEC}([\texttt{"L\_1:"}, \cdots, \texttt{"goto L\_1"}], 0)\end{aligned}$$

For the recursive version, `fact_r`, the recursive call also returns control to the top of the program, adding an extra return address to the control stack[3]

$$\begin{aligned}&\mathit{DATA}(n\hat{}\cdots) \mid \mathit{EXEC}([\texttt{"dup"}, \cdots, \texttt{"call fact\_r"}], 0)\\ &\quad\stackrel{*}{\Longrightarrow}\mathit{DATA}((n-1)\hat{}\cdots) \mid \mathit{EXEC}([\texttt{"dup"}, \cdots, \texttt{"call fact\_r"}], 0)\end{aligned}$$

[3] Corresponding to: $RETURN(\cdots)\stackrel{*}{\Longrightarrow}RETURN([fact\_r, 10]\hat{}\cdots)$

By the inductive hypotheses `fact_w` and `fact_r` are equivalent with $(n-1)$ on the data stack, and this completes the proof.

While the above proof is manually constructed, it provides the raw structure for proving equivalence between Forth programs. In general to prove equivalence between two programs we would need to demonstrate that a weak bisimulation exists between their representation in the π-calculus. Typically, it would be necessary to state a suitable abstraction over the three Forth stacks to provide the equivalence. For example, two programs achieving the same task should probably be neutral from the point of view of the control-flow and return stacks, and should have the same effect on a specified top section of the data stack.

7 Conclusions and Further Work

In this paper we have presented a formal specification of aspects of the Forth programming language, in particular the processes involved in the compilation of Forth control words. We see the contribution of this work falling into three main categories:

- *The formal specification of aspects of the Forth system.* As a stack-based machine, Forth exhibits many features similar to other stack-based languages, a technology crucial to embedded systems. We hope that a formal study of Forth constructs can act as a foundation for the study and comparison of such stack-based languages, and contribute to the verification of their safety properties.
- *A formal explication of Forth control structures.* The compilation of Forth control words, being a mixture of syntactically simple, but nonetheless structured constructs, is unique to Forth. The study of such structures can contribute to the general field of research surrounding the user of low-level control structures that facilitate the data-flow analysis process, central to, for example, bytecode verification in the JVM.
- *The use of the π-calculus in programming language specification.* As mentioned in Section 2, this approach gives a different perspective to traditional compositional semantics. Such a specification is particularly suited to languages with little syntactic structure that have to interact with a number of different data structures or devices.

We have already noted the basic similarity between Forth programs and stack-based virtual machines such as the JVM or CLR. Indeed, as we noted when comparing our semantics to those of $\mathrm{JVM}_{\mathcal{I}}$, Forth code represents a further level of abstraction, where the mode of dealing with local variables has not yet been made concrete. As such, a semantics of the core of Forth may be seen as a common foundation for stack-based virtual machines and even register-based virtual machines such as Parrot [Sugalski 2003], and this is an area we intend to investigate further. The contribution of research on Forth to the optimisation of the JVM has been well-noted [Ertl et al. 2002], and it is our hope that a similar contribution may be exacted in the semantic domain.

References

[Barr and Frank 1997] Barr, M. and Frank, B. Java: Too much for your system? *Embedded Systems Programming*, pages 24–32 (May, 1997).

[Börger and Schulte 1998] Börger, E. and Schulte, W. Defining the Java virtual machine as a platform for provably correct Java compilation. In *2nd International Symposium on Mathematical Foundations of Computer Science*, Brno, Czech Republic (August 24-28, 1998).

[ECMA-335 2001] ECMA-335. *Common Language Infrastructure (CLI).* European Computer Manufacturers Association (December, 2001).

[Ertl et al. 2002] Ertl, M. A., Gregg, D., Krall, A., and Paysan, B. vmgen – a generator of efficient virtual machine interpreters. *Software–Practice and Experience*, 32(3):265–294 (2002).

[Franz 1998] Franz, M. Open standards beyond Java: On the future of mobile code for the internet. *Journal of Universal Computer Science*, 4(5):521–532 (May, 1998).

[Gosling et al. 1996] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison Wesley (1996).

[Hennessy 1990] Hennessy, M. *The Semantics of Programming Languages.* Wiley (1990).

[ISO 1997] ISO. *ISO/IEC 15145:1997 Information technology - Programming languages - Forth.* International Standards Organisation (1997).

[Knaggs 1993] Knaggs, P. J. *Practical and Theoretical Aspects of Forth Software Development.* PhD thesis, University of Teesside (March, 1993).

[Koopman 1989] Koopman, P. J. *Stack Computers: the new wave.* Ellis Horwood (1989).

[Lindholm and Yellin 1996] Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification.* Addison Wesley (1996).

[Milner 1999] Milner, R. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press (1999).

[Moore and Leach 1970] Moore, C. H. and Leach, G. *FORTH A Language for Interactive Computing.* Mohasco Industries Inc., Amsterdam, NY (1970).

[Pierce and Turner 1997] Pierce, B. C. and Turner, D. N. Pict: a programming language based on the π-calculus. Technical report, Computer Science Department, Indiana University (1997).

[Pöial 1993] Pöial, J. Some ideas on formal specification of Forth programs. In *EuroFORTH conference on the FORTH programming language and FORTH processors*, Marianske Lazne, Czech Republic (October 15-18, 1993).

[Pöial 2003] Pöial, J. Program analysis for stack based languages. In *EuroFORTH conference on the FORTH programming language and FORTH processors*, Herefordshire, UK (October 17-19, 2003).

[Power and Sinclair 2001] Power, J. and Sinclair, D. A formal model of Forth control words in the Pi-calculus - and its animation in Pict. Technical Report Technical Report NUIM-CS-TR-2001-03, Dept. of Computer Science, National University of Ireland, Maynooth (February, 2001). Source code available from: `http://www.cs.may.ie/~jpower/Research/pi-forth`.

[Röckl and Sangiorgo 1999] Röckl, C. and Sangiorgo, D. A π-calculus process semantics of concurrent idealised ALGOL. In *Second International Conference on the Foundations of Software Science and Computation Structure*, pages 306–321, Amsterdam, The Netherlands (March 22-28, 1999).

[Sangiorgi and Walker 2001] Sangiorgi, D. and Walker, D. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press (2001).

[Schmidt 1986] Schmidt, D. A. *Denotational Semantics: a methodology for language development*. Allyn and Bacon (1986).

[Stoddart 1996] Stoddart, B. An event calculus model of the Forth programming system. In *EuroFORTH conference on the FORTH programming language and FORTH processors*, St. Petersburg, Russia (October 4-6, 1996).

[Sugalski 2003] Sugalski, D. The soul of a new virtual machine. *Linux Magazine* (April, 2003).

[Veldhuizen 1995] Veldhuizen, T. Using C++ template metaprograms. *C++ Report*, 7(4):36–43 (May, 1995).

[Watt 1991] Watt, D. A. *Programming Language Syntax and Semantics*. Prentice-Hall (1991).